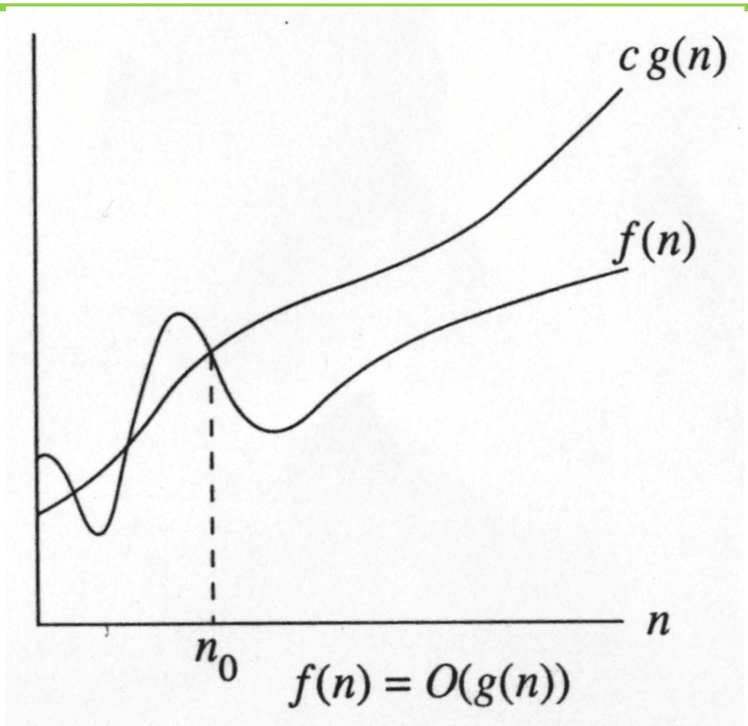
Analysis of Algorithms

Manoj Kumar
DTU, Delhi

Growth Rate



- The idea is to establish a relative order among functions for large *n*
- $\exists\, c\,,\, n_0 > 0$ such that $f(N) \leq c\; g(N)$ when $N \geq n_0$
- *f(N)* grows no faster than c *g(N)* for “large” *N*

Asymptotic notation: Big-Oh

- $f(N) = O(g(N))$
- There are positive constants c and n_0 such that

$$f(N) \leq c\, g(N) \text{ when } N \geq n_0$$

- The growth rate of $f(N)$ is *less than or equal to* the growth rate of $g(N)$
- $g(N)$ is an upper bound on $f(N)$

Big-Oh: Example

Suppose $f(n) = n^2 + 3n - 1$. We want to show that $f(n) = O(n^2)$.

$$
\begin{aligned}
f(n) &= n^2 + 3n - 1 \\
&< n^2 + 3n \quad \text{(subtraction makes things smaller so drop it)} \\
&<= n^2 + 3n^2 \quad \text{(since } n <= n^2 \text{ for all integers } n\text{)} \\
&= 4n^2
\end{aligned}
$$

$F(n) = O(n^2)$ since $f(n) <= 4n^2$ for all $n >=1$ ($C=4$, $n_0 = 1$)

Show:

$$f(n) = 2n^7 - 6n^5 + 10n^2 – 5 = O(n^7)$$

$$
\begin{aligned}
f(n) &< 2n^7 + 6n^5 + 10n^2 \\
&<= 2n^7 + 6n^7 + 10n^7 \\
&= 18n^7
\end{aligned}
$$

thus, with $C = 18$ and we have shown that $f(n) = O(n^7)$

Big-Oh: Example

Consider the sorting algorithm shown below. Find the number of instructions executed and the complexity of this algorithm.

```
1)   for (i = 1; i < n; i++) {                                       n
2)           SmallPos = i;                                           n-1
3)           Smallest = Array[SmallPos];                             n-1
4)           for (j = i+1; j <= n; j++)                              (n-1)*(n-2)/2
5)                   if (Array[j] < Smallest) {                      (n-1)*(n-2)/2
6)                           SmallPos = j;                           (n-1)*(n-2)/2
7)                           Smallest  = Array[SmallPos]             (n-1)*(n-2)/2
                     }
8)           Array[SmallPos] = Array[i];                             n-1
9)           Array[i] = Smallest;                                    n-1
     }
```

The total computing time is:

$$
\begin{aligned}
f(n) &= (n) + 4(n-1) + 4(n-1)(n-2)/2 \\
&= n + 4n - 4 + 2(n^2 - 3n + 2) \\
&= 5n - 4 + 2n^2 - 6n + 4 \\
&= 5n + 2n^2 - 6n \\
&= 2n^2 - n \\
&\leq 2n^2 \quad \text{for all } n \geq 1 \\
&= O(n^2)
\end{aligned}
$$

SAMSUNG

DTU Delhi Technological University

Big-Oh: example

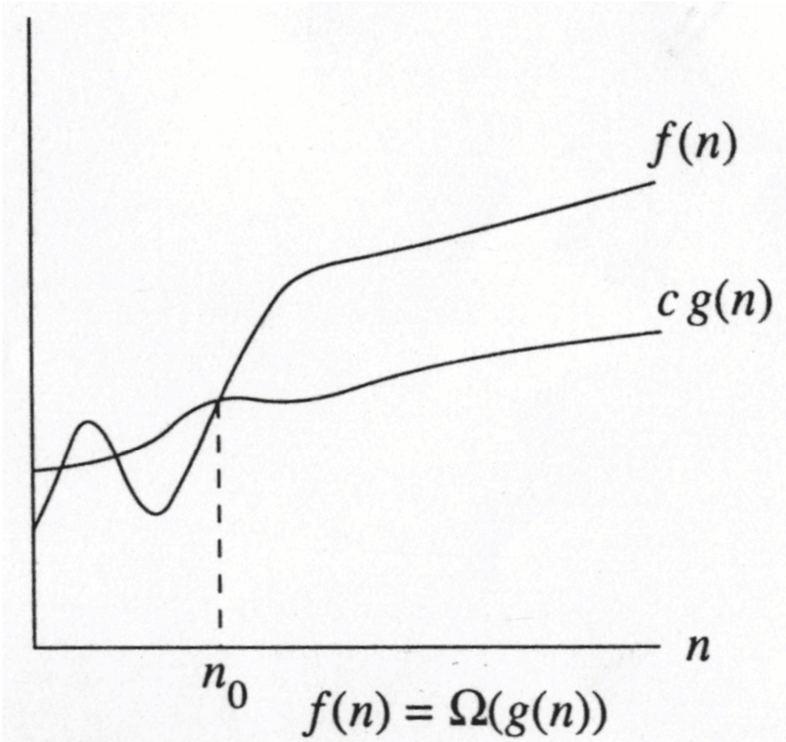
- Let $f(N) = 2N^2$. Then
  - $f(N) = O(N^4)$
  - $f(N) = O(N^3)$
  - $f(N) = O(N^2)$ (best answer, asymptotically tight)

- $O(N^2)$: reads "**order *N*-squared**" or "***Big-Oh N-squared***"

Big Oh: more examples

- $N^2 / 2 - 3N = O(N^2)$
- $1 + 4N = O(N)$
- $7N^2 + 10N + 3 = O(N^2) = O(N^3)$
- $\log_{10} N = \log_2 N / \log_2 10 = O(\log_2 N) = O(\log N)$
- $\sin N = O(1);\ 10 = O(1),\ 10^{10} = O(1)$
- $$\sum_{i=1}^{N} i \leq N \cdot N = O(N^2)$$
  $$\sum_{i=1}^{N} i^2 \leq N \cdot N^2 = O(N^3)$$
- $\log N + N = O(N)$
- $\log^k N = O(N)$ for any constant k

Big-Omega

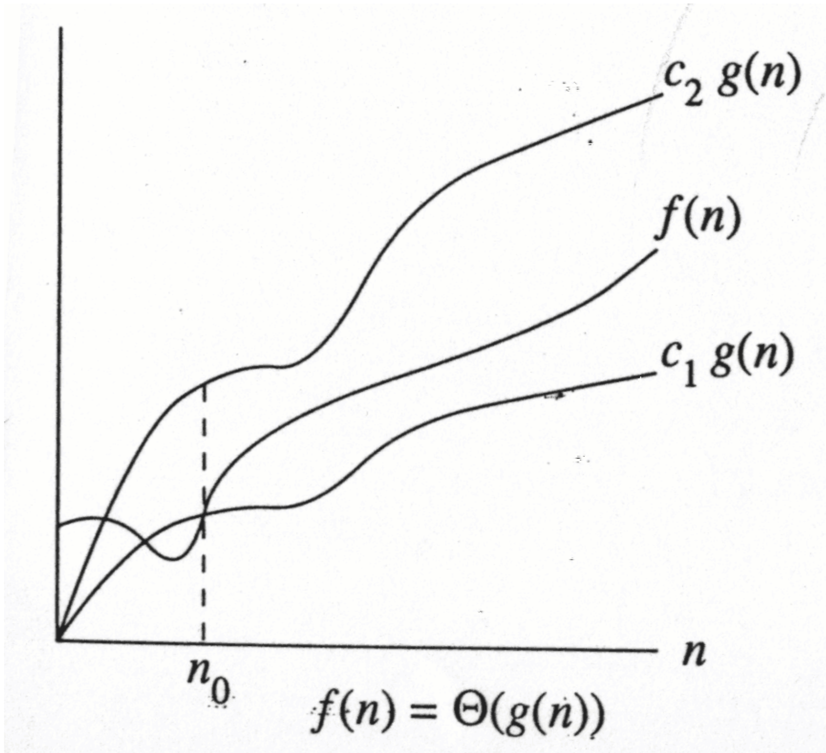


- $f(N) = \Omega(g(N))$
- $\exists\, c\, ,\, n_0 > 0$ such that $f(N) \geq c\; g(N)$ when $N \geq n_0$
- $f(N)$ grows no slower than c $g(N)$ for "large" N

Big-Omega: examples

- Let $f(N) = 2N^2$. Then
  - $f(N) = \Omega(N)$
  - $f(N) = \Omega(N^2)$ (best answer)

Big Theta



- $f(N) = \Theta(g(N))$
- $\exists\, c1, c2,\ n_0 > 0$ such that $c1\ g(N) \le f(N) \le\ c2\ g(N)$ when $N \ge n_0$
- *f(N)* grows no slower than *c1 g(N)* and no faster than *c2 g(N)* for "large" *N*
- the growth rate of *f(N) is the same as* the growth rate of *g(N)*

Big Theta

- $f(N) = \Theta(g(N))$ iff

  $f(N) = O(g(N))$ and $f(N) = \Omega(g(N))$
- The growth rate of $f(N)$ *equals* the growth rate of $g(N)$
- Example: Let $f(N)=N^2$, $g(N)=2N^2$
  - Since $f(N) = O(g(N))$ and $f(N) = \Omega(g(N))$,
  - thus $f(N) = \Theta(g(N))$.
- Big-Theta means the bound is the tightest possible.

SAMSUNG

DTU Delhi Technological University

Some Rules

- If $T(N)$ is a polynomial of degree k, then

  $T(N) = \Theta(N^k)$.

- For logarithmic functions,

  $T(\log_m N) = \Theta(\log N)$.

General Rules

- For loops
  - at most the running time of the statements inside the for-loop (including tests) times the number of iterations.
- Nested for loops

```
for (i=0;i<N; i++)
        for (j=0;j<N;j++)
                k++;
```

  - the running time of the statement multiplied by the product of the sizes of all the for-loops.
  - $O(N^2)$

General Rules

- Consecutive statements

```
for (i=0; i<N; i++)
        a[i]=0;
for (i=0; i<N; i++)
        for (j=0; j<N; j++)
                a[i]=a[i]+ a[j]+ i + j;
```

  - These just add
  - $O(N) + O(N^2) = O(N^2)$

- IF-ELSE statements

```
if (cond) then        O(1)
     S1               T1(n)
else
     S2               T2(n)
```

  - never more than the running time of the test plus the larger of the running times of S1 and S2.

    $T(n) = O(max\ (T_1(n), T_2(n))$

General Rules

Method calls

- A calls B
- B calls C
- etc.

A sequence of operations when call sequences are flattened

$$T(n) = max(T_A(n), T_B(n), T_C(n))$$

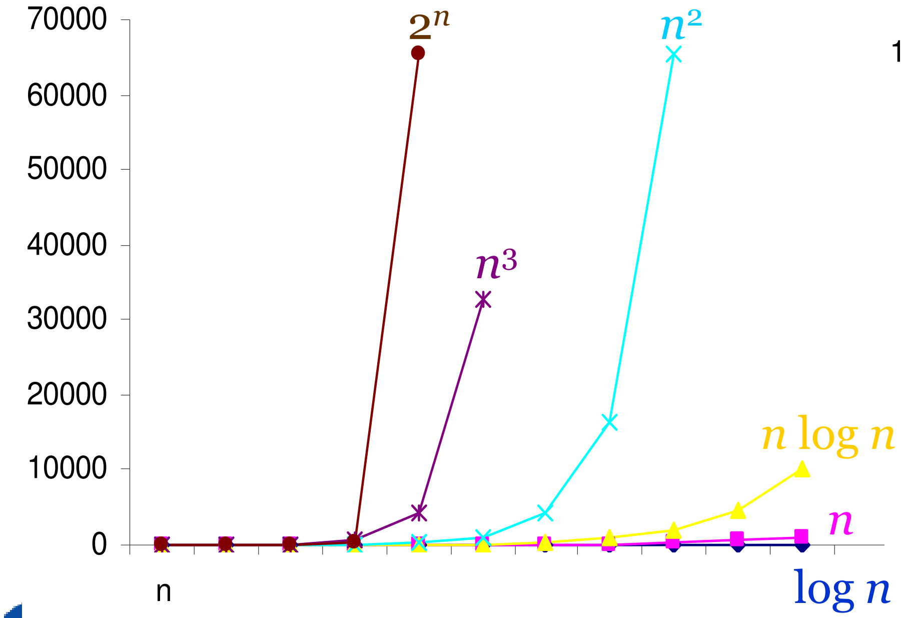
Complexity and Tractability

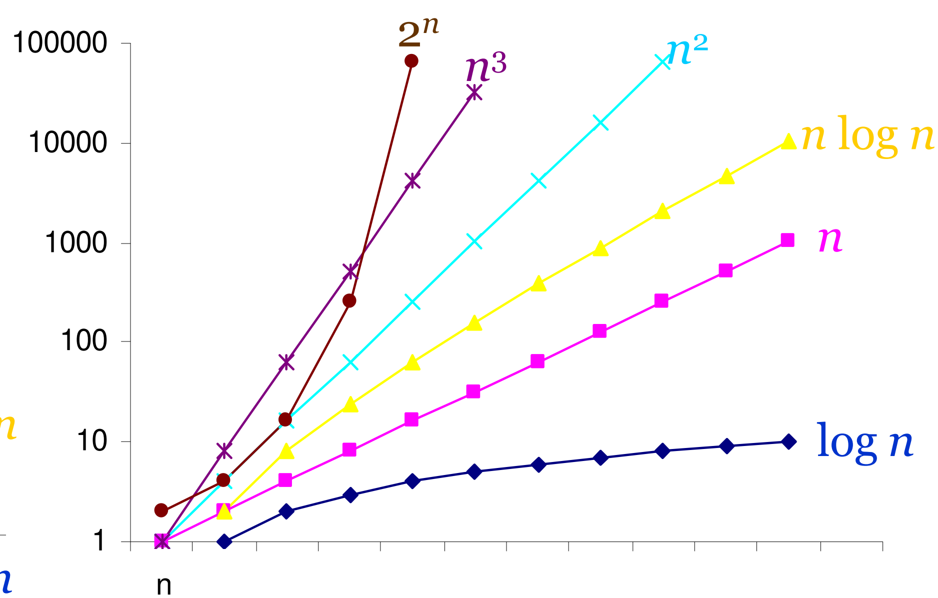
	$T(n)$							
n	n	$n \log n$	n^2	n^3	n^4	n^{10}	2^n
10	.01µs	.03µs	.1µs	1µs	10µs	10s	1µs
20	.02µs	.09µs	.4µs	8µs	160µs	2.84h	1ms
30	.03µs	.15µs	.9µs	27µs	810µs	6.83d	1s
40	.04µs	.21µs	1.6µs	64µs	2.56ms	121d	18m
50	.05µs	.28µs	2.5µs	125µs	6.25ms	3.1y	13d
100	.1µs	.66µs	10µs	1ms	100ms	3171y	4×10^{13}y
10^3	1µs	9.96µs	1ms	1s	16.67m	3.17×10^{13}y	32×10^{283}y
10^4	10µs	130µs	100ms	16.67m	115.7d	3.17×10^{23}y	
10^5	100µs	1.66ms	10s	11.57d	3171y	3.17×10^{33}y	
10^6	1ms	19.92ms	16.67m	31.71y	3.17×10^7y	3.17×10^{43}y	

Assume the computer does 1 billion ops per sec.

Complexity and Tractability

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65,536
5	32	160	1,024	32,768	4,294,967,296





Analysis of Recursive Algorithms

Recursion

A function is defined recursively if it has the following two parts

- An anchor or base case
  - The function is defined for one or more specific values of the parameter(s)
- An inductive or recursive case
  - The function's value for current parameter(s) is defined in terms of previously defined function values and/or parameter(s)

Recursion:Example

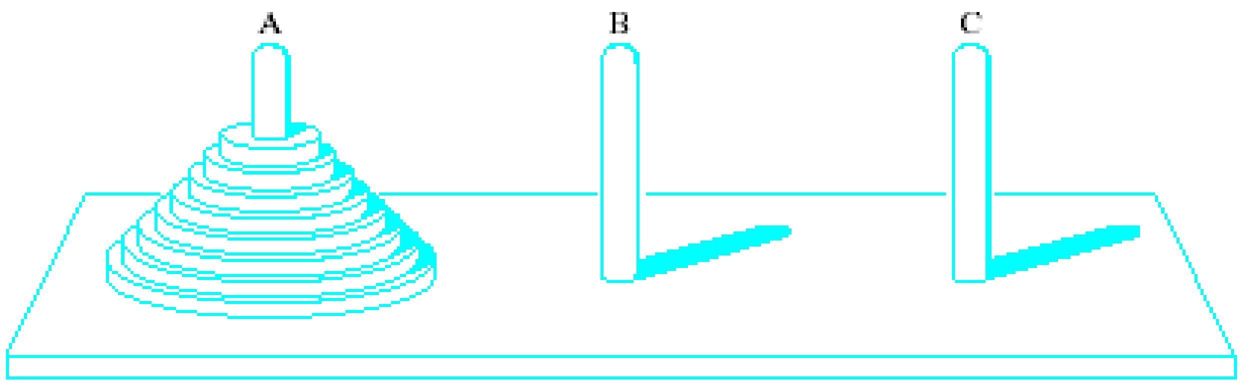
- Consider a recursive power function

```
double power (double x, unsigned n)
{   if   ( n == 0 )
            return 1.0;
     else
            return x * power (x, n-1);   }
```

- Which is the anchor?
- Which is the inductive or recursive part?
- How does the anchor keep it from going forever?
- Recurrence $T(n) = T(n-1) + O(1)$

Recursion Example: Towers of Hanoi

- Recursive algorithm especially appropriate for solution by recursion



- Task
  - Move disks from left peg to right peg
  - When disk moved, must be placed on a peg
  - Only one disk (top disk on a peg) moved at a time
  - Larger disk may never be placed on a smaller disk

Recursion Example: Towers of Hanoi

- Identify base case:
  If there is one disk move from A to C
- Inductive solution for n > 1 disks
  - Move topmost n – 1 disks from A to B, using C for temporary storage
  - Move final disk remaining on A to C
  - Move the n – 1 disk from B to C using A for temporary storage
- View code for solution,

Recursion Example: Towers of Hanoi

CODE

```
TowerOfHanoi(int n, char peg1, char peg3, char peg2)
{         // transfer n disks from peg1 to peg 3 using peg2
          if ( n==1)
                    printf(" Move disk from %c to %c\n" , peg1, peg3);
          else
          {
                    TowerOfHanoi(n-1, peg1, peg2, peg3);
                    printf("Move disk from %c to %c\n", peg1,peg3);
                    TowerOfHanoi(n-1, peg2, peg3, peg1);
          }
}
```

Recurrence: $T(n) = 2T(n-1) + 1$

Tower of Hanoi: Analysis

- Recurrence: $T(n) = 2T(n-1) + 1$

$T(1) = 1$

$T(2) = 2T(1) + 1 = 2 + 1 = 3$

$T(3) = 2T(2) + 1 = 2x3 + 1 = 7$

$T(4) = 2T(3) + 1 = 2x7 + 1 = 15 = 2^4 - 1$

…

$T(n) = 2^n - 1 = O(2^n)$

Binary Search: Recurrence

```
BINARY-SEARCH (A, lo, hi, x)
{
    if (lo > hi)                            ← constant time: c1
        return FALSE
    mid = ⌊(lo+hi)/2⌋ ;                     ← constant time: c2
    if  (x = A[mid])
        return TRUE;                        ← constant time: c3
    if ( x < A[mid] )
        BINARY-SEARCH (A, lo, mid-1, x);    ← same problem of size n/2
    if ( x > A[mid] )
        BINARY-SEARCH (A, mid+1, hi, x);    ← same problem of size n/2
}
```

- Recurrence : $T(n) = c + T(n/2)$

Solving Recurrences :ITERATION

ITERATION : Example1

$$\boldsymbol{T(n) = c + T(n/2)}$$

$$T(n) = c + T(n/2)$$
$$= c + c + T(n/4)$$
$$= c + c + c + T(n/8)$$

$$T(n/2) = c + T(n/4)$$
$$T(n/4) = c + T(n/8)$$

$$Assume\ n = 2^k$$

$$T(n) = \underbrace{c + c + \ldots + c}_{\text{k times}} + T(1)$$
$$= c \lg n + T(1)$$
$$= \Theta(\lg n)$$

Solving Recurrence: ITERATION

- Example2

$$\boldsymbol{T(n) = n + 2T(n/2)}$$

$$
\begin{aligned}
T(n) &= n + 2T(n/2) \\
&= n + 2(n/2 + 2T(n/4)) \\
&= n + n + 4T(n/4) \\
&= n + n + 4(n/4 + 2T(n/8)) \\
&= n + n + n + 8T(n/8) \\
\ldots &= in + 2^iT(n/2^i) \\
&= kn + 2^kT(1) \\
&= n \lg n + nT(1) = \Theta(n \lg n)
\end{aligned}
$$

Substitution method

1. ***Guess*** the form of the solution.
2. ***Verify*** by induction.
3. ***Solve*** for constants.

- Apply only when it is easy to guess the form of answer

Example: $T(n) = 4T(n/2) + 100n$

- [Assume that $T(1) = \Theta(1)$.]
- Guess $O(n^3)$. (Prove O and Ω separately.)
- Assume that $T(k) \le ck^3$ for $k < n$.
- Prove $T(n) \le cn^3$ by induction.

SAMSUNG

DTU Delhi Technological University

Example of substitution

$$T(n) = 4T(n/2) + 100n$$
$$\leq 4c(n/2)^3 + 100n$$
$$= (c/2)n^3 + 100n$$
$$= cn^3 - ((c/2)n^3 - 100n) \quad \leftarrow \textit{desired – residual}$$
$$\leq cn^3 \quad \leftarrow \textit{desired}$$

whenever $(c/2)n^3 - 100n \geq 0$, for example, if $c \geq 200$ and $n \geq 1$.

↖ *residual*

Example (continued)

- We must also handle the initial conditions, that is, ground the induction with base cases.
- ***Base:*** $T(n) = \Theta(1)$ for all $n < n_0$, where n_0 is a suitable constant.
- For $1 \leq n < n_0$, we have "$\Theta(1)$" $\leq cn^3$, if we pick c big enough.

- ***This bound is not tight!***

A tighter upper bound?

- We shall prove that $T(n) = O(n^2)$.
- Assume that $T(k) \le ck^2$ for $k < n$:

$$T(n) = 4T(n/2) + 100n$$

$$\le cn^2 + 100n$$

$$\le cn^2$$

- for ***no*** choice of $c > 0$. Lose!

A tighter upper bound!

IDEA: Strengthen the inductive hypothesis.

- ***Subtract*** a low-order term.
  - *Inductive hypothesis*: $T(k) \le c_1k^2 - c_2k$ for $k < n$.

  - T(n) =4T(n/2) + 100 n

$$\le 4(c_1(n/2)^2 - c_2(n/2)) + 100n$$
$$= c_1n^2 - 2c_2\, n + 100n$$
$$= c_1n^2 - c_2\, n - (c_2\, n - 100\, n)$$
$$\le c_1n^2 - c_2\, n \quad \text{if } c_2 > 100$$

Pick c_1 big enough to handle the initial conditions.

SAMSUNG

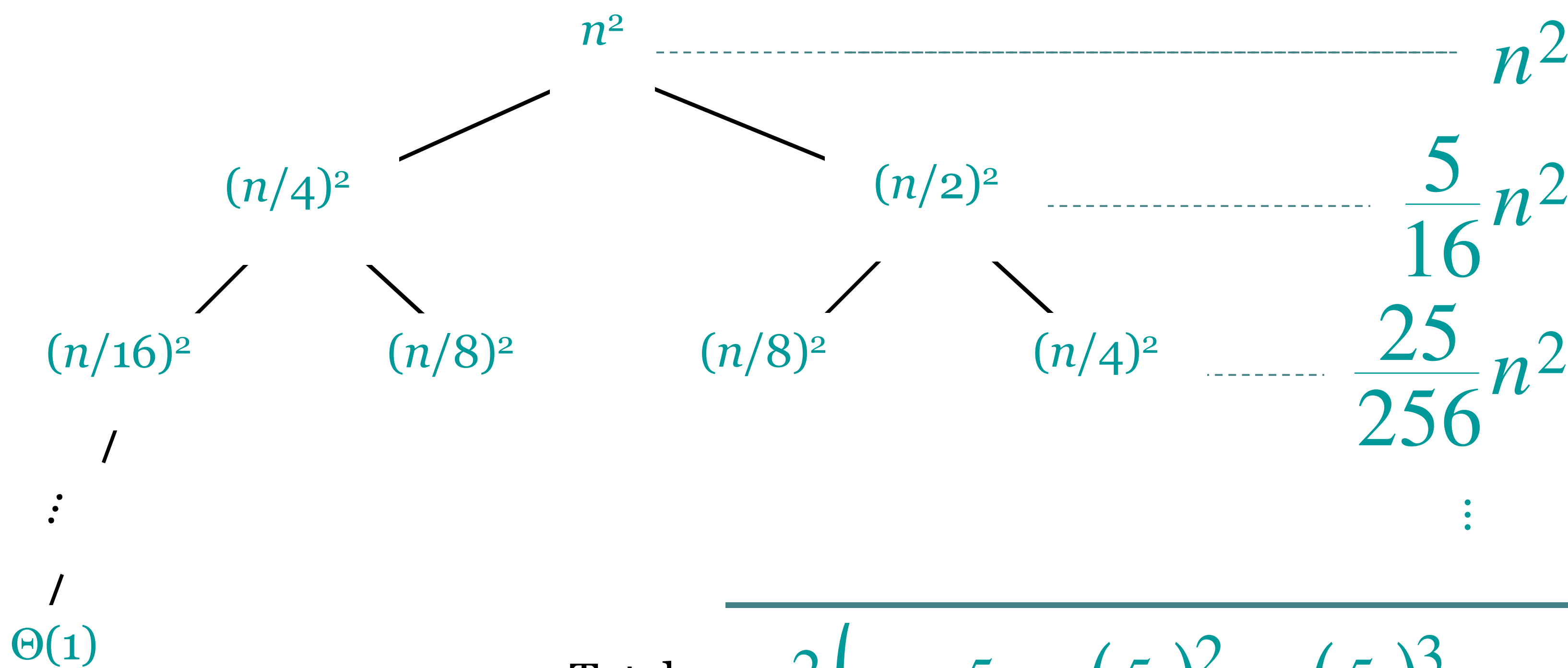
DTU Delhi Technological University

Recursion-tree method

- A recursion tree models the costs (time) of a recursive execution of an algorithm.
- The recursion tree method is good for generating guesses for the substitution method.
- The recursion-tree method can be unreliable, just like any method that uses ellipses (…).
- The recursion-tree method promotes intuition, however.

Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



$$\text{Total} = n^2\left(1 + \frac{5}{16} + \left(\frac{5}{16}\right)^2 + \left(\frac{5}{16}\right)^3 + \cdots\right)$$

$= \Theta(n^2)$ *geometric series*

The master method

- The master method applies to recurrences of the form
  - $T(n) = a\,T(n/b) + f(n)$,
- where $a \geq 1$, $b > 1$, and f is asymptotically positive.

Master method: Case1

Compare $f(n)$ with $n^{\log_b a}$:

1. $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$.
   - $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by an n^{ε} factor).

- ***Solution:*** $T(n) = \Theta(n^{\log_b a})$.

Masters theorem: Case 2

Compare $f(n)$ with $n^{\log_b a}$:

2. $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some constant $k \geq 0$.
   - $f(n)$ and $n^{\log_b a}$ grow at similar rates.
   - ***Solution:*** $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

Masters theorem: Case 3

Compare $f(n)$ with $n^{\log_b a}$:

3. $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$.
   - $f(n)$ grows polynomially faster than $n^{\log_b a}$ (by an n^{ε} factor), ***and*** $f(n)$ satisfies the ***regularity condition*** that $af(n/b) \leq cf(n)$ for some constant $c < 1$.

   Solution: $T(n) = \Theta(f(n))$.

Examples

Ex. $T(n) = 4T(n/2) + n$

$a = 4$, $b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n$.

CASE1: $f(n) = O(n^{2-\varepsilon})$ for $\varepsilon = 1$.

$\therefore T(n) = \Theta(n^2)$.

Ex. $T(n) = 4T(n/2) + n^2$

$a = 4$, $b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n^2$.

CASE2: $f(n) = \Theta(n^2 \lg^0 n)$, that is, $k = 0$.

$\therefore T(n) = \Theta(n^2 \lg n)$.

Example

Ex. $T(n) = 4T(n/2) + n^3$

$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n^3.$

CASE3: $f(n) = \Omega(n^{2+\varepsilon})$ for $\varepsilon = 1$

and $4(cn/2)^3 \le cn^3$ (reg. cond.) for $c = 1/2$.

$\therefore\ T(n) = \Theta(n^3)$.

Ex. $T(n) = 4T(n/2) + n^2/\lg n$

$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n^2/\lg n.$

Master method does not apply. In particular, for every constant $\varepsilon > 0$, we have $n^\varepsilon = \omega(\lg n)$.

Amortized analysis

- An *amortized analysis is any strategy for analyzing a sequence of operations to show that the average cost per operation is small, even though a single operation within the sequence might be expensive.*

- *Amortized analysis differs from average case analysis in that probability is not involved.*

- An amortized analysis guarantees the average performance of each operation in the*worst case*.

Types of amortized analysis

- Three common amortization arguments:
  - The ***aggregate method,***
  - The ***accounting method,***
  - The ***potential method.***

The *aggregate method*

- We show that for all n, if a sequence of n operations takes worst-case time T(n) in total, then amortized cost per operation is therefore T(n)/n.
- Example : incrementing a binary counter.

Binary Counter

- Consider a k-bit binary counter that counts upwards from 0. We use an array A[0..k-1] of bits.
- A binary number stored in the counter has its lowest bit in A[0] and its highest bit in A[k-1], so that

$$x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$$

- To add 1 to the value in the counter, we use following procedure

```
INCREMENT(A)
1. i ← 0
2. while i < length[A] and A[i] = 1
3.      do A[i] ← 0          ▷ reset a bit
4.         i ← i + 1
5. if i < length[A]
6.      then A[i] ← 1        ▷ set a bit
```

Ctr	A[4]	A[3]	A[2]	A[1]	A[0]	*Cost*
0	0	0	0	0	0	*0*
1	0	0	0	0	1	*1*
2	0	0	0	1	0	*3*
3	0	0	0	1	1	*4*
4	0	0	1	0	0	*7*
5	0	0	1	0	1	*8*
6	0	0	1	1	0	*10*
7	0	0	1	1	1	*11*
8	0	1	0	0	0	*15*
9	0	1	0	0	1	*16*
10	0	1	0	1	0	*18*
11	0	1	0	1	1	*19*
12	0	1	1	0	0	*22*
13	0	1	1	0	1	*23*
14	0	1	1	1	0	*25*
15	0	1	1	1	1	*26*
16	1	0	0	0	0	*31*

Worst-case analysis

- Consider a sequence of n increments. The worst-case time to execute one increment is $\Theta(k)$. Therefore, the worst-case time for n increments is $n \cdot \Theta(k) = \Theta(n \cdot k)$.
- **WRONG!** In fact, the worst-case cost for n increments is only $\Theta(n) \ll \Theta(n \cdot k)$.
- Let's see why.

Tighter analysis

Ctr	A[4]	A[3]	A[2]	A[1]	A[0]	*Cost*
0	0	0	0	0	0	*0*
1	0	0	0	0	1	*1*
2	0	0	0	1	0	*3*
3	0	0	0	1	1	*4*
4	0	0	1	0	0	*7*
5	0	0	1	0	1	*8*
6	0	0	1	1	0	*10*
7	0	0	1	1	1	*11*
8	0	1	0	0	0	*15*
9	0	1	0	0	1	*16*
10	0	1	0	1	0	*18*
11	0	1	0	1	1	*19*
12	0	1	1	0	0	*22*
13	0	1	1	0	1	*23*
14	0	1	1	1	0	*25*
15	0	1	1	1	1	*26*
16	1	0	0	0	0	*31*

Total cost of n operations

A[0] flipped every op n

A[1] flipped every 2 ops $n/2$

A[2] flipped every 4 ops $n/2^2$

A[3] flipped every 8 ops $n/2^3$

… … … … …

A[i] flipped every 2^i ops $n/2^i$

Tighter analysis…

Cost of n increments $= \sum_{i=1}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor$

$$< n \sum_{i=1}^{\infty} \frac{1}{2^i} = 2n$$

$$= \Theta(n)$$

Thus, the average cost of each increment operation is $\Theta(n)/n = \Theta(1)$.

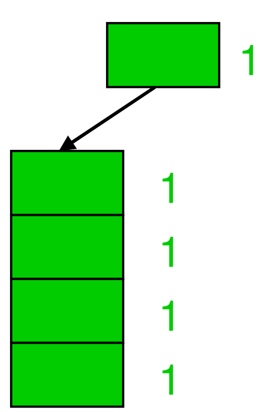
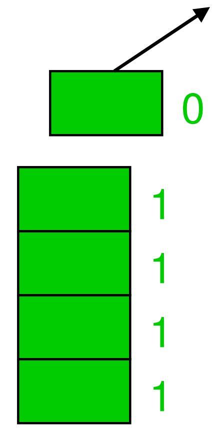
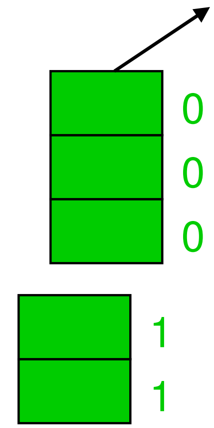
SAMSUNG

DTU Delhi Technological University

Accounting method

- We assign differing charges to different operations, with some operations charged more or less than they actually cost.
- The amount we charge an operation is called ***amortized cost***.
- When an operation's amortized cost exceeds its actual cost, the difference is called credit.
- Credit can be used later to pay for operations whose amortized cost is less than their actual cost.

A Simple Example: Accounting method

3 ops:			
	Push(S,x)	Pop(S)	Multi-pop(S,k)
•Amortized cost:	2	0	0
•Actual cost:	1	1	$\min(\lvert S\rvert,k)$

Push(S,x) pays for possible later pop of x.

Stack Example: Accounting Method

- When pushing an object, pay \$2
  - \$1 pays for the push
  - \$1 is prepayment for it being popped by either pop or Multipop
  - Since each object has \$1, which is credit, the credit can never go negative
  - Therefore, total amortized cost = O(n), is an upper bound on total actual cost

Accounting analysis of INCREMENT

- Charge an amortized cost of \$2 every time a bit is set from 0 to 1
  - \$1 pays for the actual bit setting.
  - \$1 is stored for later re-setting (from 1 to 0).
- At any point, every 1 bit in the counter has \$1 on it… that pays for resetting it. (reset is "*free*")

Example:

0 0 0 1[\$1] 0 1[\$1] 0

0 0 0 1[\$1] 0 1[\$1] 1[\$1] **Cost = \$2**

0 0 0 1[\$1] 1[\$1] 0 0 **Cost = \$2**

Incrementing a Binary Counter

```
INCREMENT(A)
1.  i ← 0
2.  while i < length[A] and A[i] = 1
3.      do A[i] ← 0          ▷ reset a bit
4.         i ← i + 1
5.  if i < length[A]
6.      then A[i] ← 1        ▷ set a bit
```

- When Incrementing,
  - Amortized cost for line 3 = \$0
  - Amortized cost for line 6 = \$2
- Amortized cost for INCREMENT(A) = \$2
- Amortized cost for n INCREMENT(A) = \$2n =$O(n)$

The potential method

- Represent prepaid work as ***"potential energy"*** or ***"potential"***, that can be released to pay for future operations.
- Potential is associated with the data structure as a whole, rather than with specific object within the data structure.

The potential method

- Start with an initial data structure D_0.
- Operation i transforms D_{i-1} to D_i.
- The actual cost of operation i is c_i.
- Define a ***potential function*** $\Phi : \{D_i\} \to \mathsf{R}$,
  such that $\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i.
- The ***amortized cost*** $\hat{c}_i$ with respect to Φ is defined to be $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$.

i.e. Amortized cost = actual cost + increase in potential due to operation.

Understanding potential

$$\hat{c}_i = c_i + \underbrace{\Phi(D_i) - \Phi(D_{i-1})}_{\textit{potential difference } \Delta\Phi_i}$$

- If $\Delta\Phi_i > 0$, then $\hat{c}_i > c_i$. Operation i stores work in the data structure for later use.
- If $\Delta\Phi_i < 0$, then $\hat{c}_i < c_i$. The data structure delivers up stored work to help pay for operation i.

Amortized costs bound the true costs

The total amortized cost of n operations is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1}) \right)$$

$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

$$\geq \sum_{i=1}^{n} c_i$$

since $\Phi(D_n) \geq 0$ and $\Phi(D_0) = 0$.

Stack Example: Potential

Define: $\phi(D_i)$ = #items in stack Thus, $\phi(D_0)=0$.

Plug in for operations:

Push: $\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1})$
$= 1 + j - (j-1)$
$= 2$

Pop: $\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1})$
$= 1 + (j-1) - j$
$= 0$

Multi-pop: $\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1})$
$= k' + (j-k') - j$ $k'=\min(|S|,k)$
$= 0$

Potential analysis of INCREMENT

Define the potential of the counter after the i^{th} operation by $\Phi(D_i) = b_i$, the number of 1's in the counter after the i^{th} operation.

Note:

- $\Phi(D_0) = 0$,
- $\Phi(D_i) \geq 0$ for all i.

Example:

0 0 0 1 0 1 0 $\Phi = 2$

(0 0 0 $1^{\$1}$ 0 $1^{\$1}$ 0 Accounting method)

Potential analysis of INCREMENT

Assume ith INCREMENT resets t_i bits (in line 3).
Actual cost $c_i = (t_i + 1)$
Number of 1's after ith operation: $b_i = b_{i-1} - t_i + 1$

The amortized cost of the ith INCREMENT is

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= (t_i + 1) + b_i - b_{i-1} \\ &= (t_i + 1) + (1 - t_i) \\ &= 2\end{aligned}$$

Therefore, n INCREMENTS cost $\Theta(n)$ in the worst case.

Disjoint Sets

Manoj Kumar
DTU, Delhi

Disjoint Sets

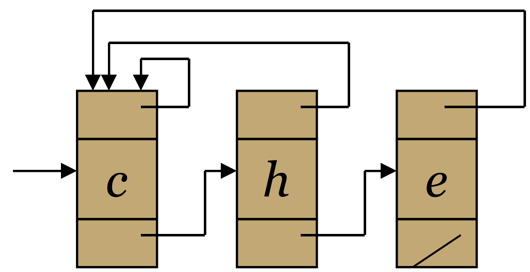
- A disjoint set data structure maintains a collection $S=\{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets.
- Each set is identified by a representative, which is some member of the set.
- Supports following operations:
  - **MAKE_SET(x):** creates a new set whose only member is pointed by x.
  - **UNION(x,y):** unite the dynamic sets that contains x and y, say S_x and S_y.
  - **FIND_SET(x):** returns a pointer to the representative of the set containing x.

Linked-List Implementation

- Each set as a linked-list, with head and tail, and each node contains value, next node pointer and back-to-representative pointer.

Pointer to representative node
Value
Pointer to other member

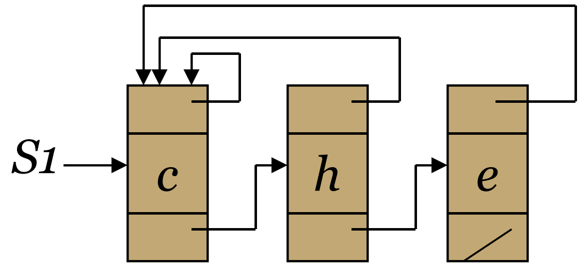
Node structure



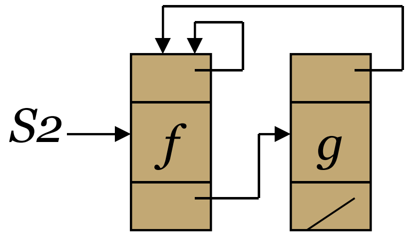
Set $S_1=\{c,h,e\}$

Linked-List for two sets

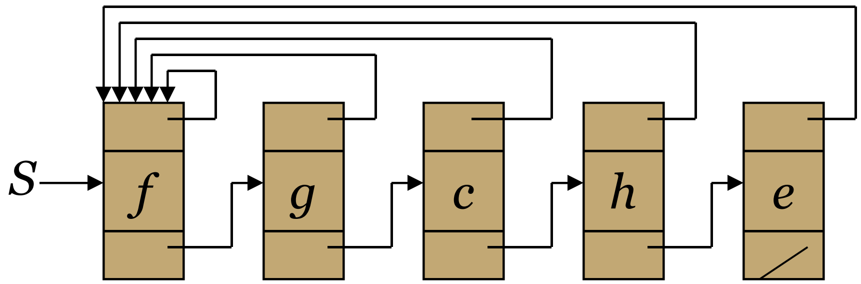
Set S1={c,h,e}



Set S2={f, g}

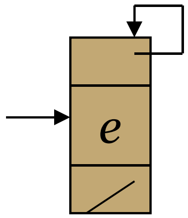


UNION of
two Sets
S=S1 U S2

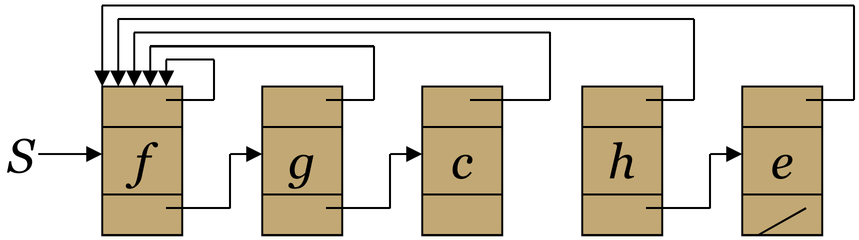


Analysis

- MAKE_SET(x) takes *O(1)* time: create a new linked list whose only object is x.



- FIND_SET(x) takes *O(1)* time: return the pointer from x back to the representative.



Union

- A simple implementation: UNION(x,y) just appends x to the end of y, updates all back-to-representative pointers in x to the head of y.
- Each UNION takes time linear in the x's length.

Union: amortized cost

- Consider sequence of m operations. m=n+q

  where q=n-1
- Let we have objects $x_1, x_2, \ldots x_n$.
- We execute n MAKE-SET(x_i) operations (O(1) each) followed by q= n-1 UNION
  - UNION(x_1, x_2), $O(1)$,
  - UNION(x_2, x_3), $O(2)$,
  - …..
  - UNION(x_{n-1}, x_n), $O(n-1) = O(q)$
- The UNIONs cost $1+2+\ldots+q=\Theta(q^2)$
- So total time spent is $\Theta(n + q^2)$, which is $\Theta(m^2)$, since n = $\Theta(m)$, and q = $\Theta(m)$.
- Thus on average, each operation require $\Theta(m^2)/m = \Theta(m)$ time, that is the amortized time of one operation.

Weighted Union

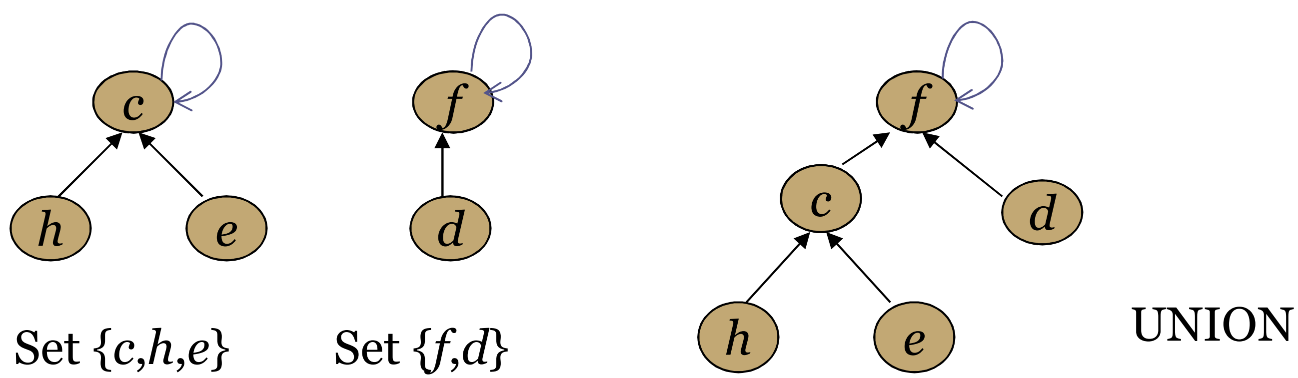
- If we are appending longer list onto a shorter list; we must update the pointer to the representative for each member of the longer list.
- Suppose each representative node also stores length of list. This can be easily maintained.
- Weighted Union: we always append smaller list onto the longer list, with ties broken arbitrarily.

Weighted Union: analysis

- Result: a sequence of *m* MAKE-SET, UNION, FIND-SET operations, *n* of which are MAKE-SET operations, the running time is $O(m+n \lg n)$. Why???
- Count the number of updates to back-to-representative pointer for any *x* in a set of *n* elements. Consider that each time, the UNION will at least double the length of united set, it will take at most ***lg n*** UNIONS to unite *n* elements. So each *x*'s back-to-representative pointer can be updated at most *lg n* times. There are n objects so all Union operations taking *n lg n* time.
- The UNION operation can stil take $\Omega(m)$ time if both sets have m elements.

Disjoint-Set Implementation: Forests

- Rooted trees, each tree is a set, root is the representative. Each node points to its parent. Root points to itself.

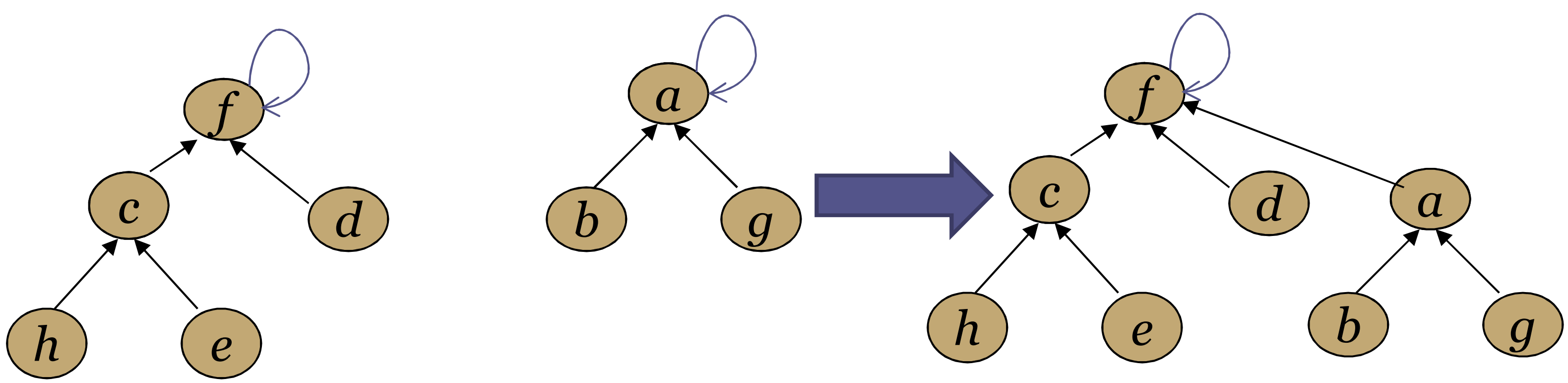


Straightforward Solution

- Three operations
  - MAKE-SET(x): create a tree containing x. $O(1)$
  - FIND-SET(x): follow the chain of parent pointers until to the root. O(height of x's tree)
  - UNION(x,y): let the root of one tree point to the root of the other. $O(1)$
- It is possible that n-1 UNIONs results in a tree of height n-1. (just a linear chain of n nodes).
- So n FIND-SET operations will cost $O(n^2)$.

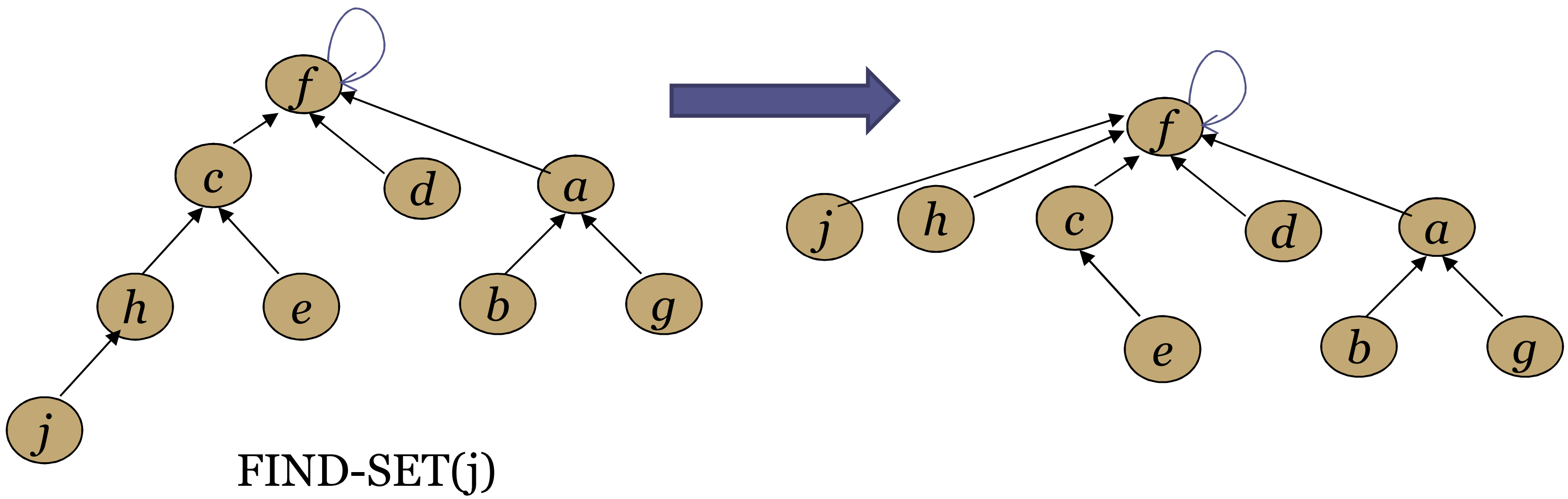
Union by Rank

- **Union by Rank**: Each node is associated with a rank, which is the upper bound on the height of the node (i.e., the height of subtree rooted at the node), then when UNION, let the root with smaller rank point to the root with larger rank.

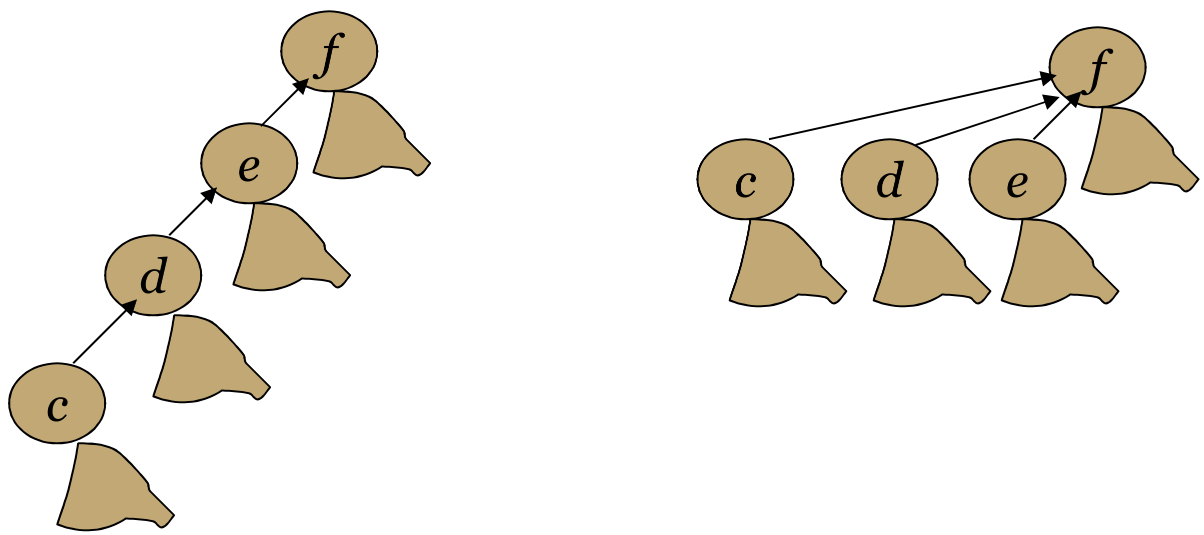


Path Compression

- **Path Compression**: used in FIND-SET(*x*) operation, make each node in the path from *x* to the root directly point to the root. Thus reduce the tree height.



Path Compression



Algorithm for Disjoint-Set Forest

MAKE-SET(x)
1. $p[x] \leftarrow x$
2. $rank[x] \leftarrow 0$

UNION(x,y)
1. LINK(FIND-SET(x),FIND-SET(y))

LINK(x,y)
1. **if** $rank[x] > rank[y]$
2. **then** $p[y] \leftarrow x$
3. **else** $p[x] \leftarrow y$
4. **if** $rank[x] = rank[y]$
5. **then** $rank[y]$++

FIND-SET(x)
1. **if** $x \neq p[x]$
2. **then** $p[x] \leftarrow$ FIND-SET($p[x]$)
3. **return** $p[x]$

Running time

- Total m operations.
- n MAKE-SET,
- At most $n-1$ UNION, and
- f FIND-SET operations
- If we use only Union by Rank, Worst case running time
  - $\Theta(n + f \log_{(1+f/n)} n)$ if $f \geq n$,
  - $\Theta(n + f \lg n)$ if $f < n$.
- If we use both Union by Rank and path compression, Worst case running time is $O(m\alpha(m,n))$ where $\alpha(m,n)$ is inverse of Ackermann's function.
- $\alpha(m,n) \leq 4$ → $O(m)$ running time → $O(m)/m = O(1)$ per operation.